Under consideration for publication in Formal Aspects of Computing

On modelling recursive calls and callbacks with two variants of Unified Modelling Language state diagrams

Jennifer Tenzer and Perdita Stevens

Laboratory for Foundations of Computer Science
School of Informatics
University of Edinburgh

Abstract. An important use of the Unified Modelling Language (UML) is modelling *synchronous* object-oriented software systems. State diagrams are used to model interesting object behaviour, including method invocation. However, almost all previous work formalising state diagrams has assumed *asynchronous* communication. We show that UML's "run to completion" semantics leads to anomalous behaviour in the synchronous case, and in particular that it is not possible to model recursive calls, in which an object receives a second synchronous message whilst still in the process of reacting to the first. We propose a solution using state diagrams in two complementary ways.

Keywords: UML, object-oriented modelling, protocol state machines, state machines, recursion, callbacks

1. Introduction

The Unified Modelling Language [OMG03, OMG04] has been widely adopted as a standard language for modelling the design of (software) systems. One diagram type within UML is the state diagram, an object-oriented adaptation of Harel statecharts.

The use to which state diagrams are put varies with the type of project and the modeller's preferences, but a typical use is as follows. The modeller decides that some or all of the classes which are to appear in the system should be modelled with state diagrams; typically, classes which are perceived to have "interesting" state change behaviour will be so modelled, whereas those which are considered to be stateless or almost so will not be. For a given class, the modeller identifies the abstract states, which will be represented as *states* in the state diagram. This involves deciding which aspects of state are interesting, in that they may affect behaviour; it can be seen as choosing an equivalence relation on the set of (concrete, fully-detailed) possible states of objects of the class. In parallel, s/he considers which *events* may happen to an object of this

Correspondence and offprint requests to: Perdita Stevens, School of Informatics, University of Edinburgh, JCMB, King's Buildings, Mayfield Road, Edinburgh EH9 3JZ, UK. email: perdita@inf.ed.ac.uk

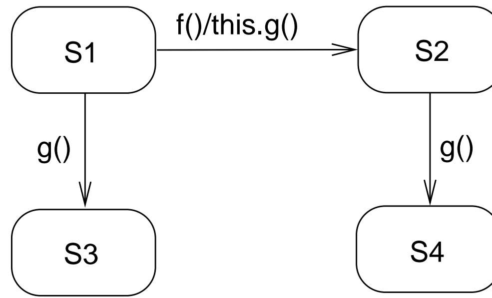

Fig. 1. Simple problem situation

class, and what effect those events have on the abstract state of the object. In the sequential single-threaded systems which are the concern of this paper, a typical event is the receipt of a message which requests the synchronous invocation of an operation. The modeller may record that certain state transitions happen only in particular circumstances; that is, s/he may add *guards* to the transitions. Finally, s/he may record how an object reacts to a given event happening in a given state by showing *actions* on the transitions (and/or within the states, but for simplicity we omit that possibility here). Typically, the modeller will not record every detail of the object's reaction, since that might involve placing the whole of an eventual method implementation as annotation on a transition in a diagram. A common compromise is to show only the messages which the object may *send* as part of its reaction to an event such as receiving a message, but not to show internal computation such as variable assignments. (Messages which an object sends directly to itself may or may not be regarded as internal computation.)

Details vary, but essentially this process is recommended in many reputable sources, including for example Booch et al.'s *UML User Guide* [BRJ98] and the second author's own *Using UML*[wRPon]. We emphasise that although a state diagram represents "all behaviours" of an object, in some appropriate sense, this does not mean that it represents *all behaviour* of the object in the sense of including all the information that will be represented in its code.

Now, much of the pre-UML expertise in using state machines was acquired in the context of event-based systems, where typical messages are asynchronous, and almost all work formalising UML state diagrams has built on this foundation and assumed asynchronous communication. In mainstream software development, however, synchronous communication is more common. In this paper we demonstrate that this leads to a problem with the UML semantics, and we propose a solution which is compatible with the intentions of the UML language designers.

Let us give a small example in which following the UML semantics. The single state diagram is shown in Figure 1. Suppose an object represented by this diagram is in state $S1$ and receives message $f()$, which causes the object to send itself the message $g()$.[1] What should the resulting state of the object be?

According to the UML semantics, there is no answer to this question: the machine deadlocks. If we consider the diagram as a complete machine specification, this is reasonable. However, this does not accord with intuition, or with the standardly suggested way of using state diagrams described above. If the designer's idea is that the implementation of $f()$ will involve the invocation of $g()$ (recorded in the diagram) and also some other computation, such as assignments to attributes (not recorded in the diagram, again following standard practice), the diagram above is exactly what will be produced, and the designer will expect the resulting state of the object to be $S2$. This is shown by the fact that the head of the sole arrow labelled f is pointing at state $S2$. It is possible that the object's state, beginning in $S1$, is changed to $S2$ by some unmodelled computation (perhaps an assignment to one of the object's attributes, whose value determines which state of the state diagram the object is in), then to $S4$ by the execution of g, then (back) to $S2$ by

[1] More precisely, $f()$ is a *call event* and $g()$ a *call action* in the transition from $S1$ to $S2$. Call events are caused by call actions and are distinct from them [OMG03] (2-142). In our example $g()$ on the transitions from $S1$ to $S3$ and $S2$ to $S4$ is a call event which is caused by the corresponding call action. Both call events and call actions are associated with an operation in UML, i.e. call events and actions together model invocations of the operation of the object. In UML2.0, the metamodel representing this situation is slightly different, but the differences do not affect our point.

further unmodelled computation (such as a second assignment to the same attribute). Alternatively, it may be the other g transition which is used, or which one is used may depend on the execution context. This underspecification is normal. Crucially, the designer does not intend the state diagram to be a complete machine: it constitutes a loose specification of the system the designer has in mind.

The fundamental problem seems to be that UML – like the UML community – is ambivalent about whether its state diagrams are intended to be *machines*, capable of being executed, or loose specifications, constraining a later implementation.[2] The UML semantics strongly suggests the former, but this does not always accord with how UML is used. In particular, when state diagrams are used to model synchronous message passing between objects in a sequential single-threaded system UML's run-to-completion semantics causes anomalies. Situations as shown above can be modelled by UML sequence diagrams and implemented in an object oriented programming language although the execution of corresponding UML state machines would result in a deadlock according to the default UML run-to-completion semantics (see example in Section 2.1). It is interesting to notice that Harel and Gery [HG97] were aware that recursive operation calls are problematic but apparently considered them unimportant. They wrote:

> ...when the client's statechart invokes another object's operation, its execution freezes in midtransition, and the thread of control is passed to the called object. Clearly, this might continue, with the called object calling others, and so on. However, a cycle of invocations leading back to the same object instance is illegal, and an attempt to execute it will abort.

In the context of synchronous object oriented systems, we do not consider that the problem can be so easily dismissed. In object oriented design recursive calls occur frequently: for example, whenever any method is recursive, or when the Visitor or Observer pattern is used. Moreover, it is not trivial for the designer to avoid cases where this problem will arise. Several state diagrams may be involved, perhaps developed by different people at different times. If a tool, adhering strictly to the UML standard, refused to accept or process such a collection of state diagrams, the result could be serious confusion and loss of confidence in the tool and/or in UML itself.

In this paper we discuss the problem and propose a solution, drawing on both the draft UML2.0 standard and relevant theoretical work [AEY01, GLS$^+$01].

A preliminary, much shorter report of this work was made in [TS03]. The main additions here are: consideration of how the adaptations of UML proposed fit into the UML1.5 metamodel; the addition of postconditions on protocol transitions; allowing new objects to be created; an informal discussion of incomplete method descriptions and how they fit into the approach; and the use of an example which is drawn from a real-world application. We also include further discussion of the impact of the move to UML2.0.

1.1. Note on standards and terminology

This paper is based on UML1.5, the current formally approved standard at the time of writing (March 2005). It also, as mentioned above, draws on the new UML2.0 standard, currently proceeding through OMG approval. In Section 7 we discuss the impact of the adoption of UML2.0. The reader is assumed to be familiar with UML; it is important to note that the definition of UML is the OMG standard [OMG03], not what is contained in any UML book.

In sequence diagrams, we show expressions on return arrows to indicate the value returned – this is common, harmless and permitted in UML2.0, but not actually described in [OMG03].

We make several simplifying assumptions. We only consider sequential systems, so our state diagrams are assumed not to make use of concurrent substates. We treat rolenames identically with attributes; for example, our attribute environments include rolenames. Such an attribute has a class type, and its value will be an object identifier. For convenience, we adopt the convention that the set of attributes of a class includes all attributes that can be derived from attributes of associated objects.

For method and object names we use sans serif style in this paper. Class names always start with a capital letter and are shown in normal style.

[2] Even the terminology in the standard shows this tension: the terms state diagram, statechart diagram, statechart and state machine are not always used consistently (compare for example [OMG03] (3-136) and (3-140)). In this paper we use "state diagram" as a general term, reserving "state machine" for an executable version.

1.2. Structure of the paper

The paper is structured as follows. In Section 2 we explain the problem with the UML's current understanding of state diagrams. In Section 3 we introduce our solution, making use of two kinds of state diagrams. An embedding of these diagrams into the UML1.5 metamodel is presented in Section 4; this section is included for the reader who is familiar with the UML metamodel, and may safely be omitted by other readers. Section 5 formalises these diagrams and defines a suitable notion of consistency. Section 6 discusses work in progress on incomplete method descriptions with respect to one of our new diagram types. Section 7 discusses the impact of the move to UML2. In Section 8 we revisit the example introduced above; Section 9 discusses related work, and Section 10 concludes.

2. State diagrams with recursive calls in UML

According to run-to-completion semantics, the action on a transition must have been completed before the transition is finished. If the action involves an object o of the class modelled by the state diagram making a call to another object p (and perhaps using the result of that call in some calculation), the action, and therefore the transition, does not complete until the call has been received by p, processed, and the result returned to o.

However, as soon as we consider the case that o and p might be the same object (recursion) or that part of p's reaction to the message from o might be to send o a new message (callback), it becomes clear that we cannot model situations with recursion or callbacks with UML state diagrams in which call events and actions involving calls are recorded on transitions.

In the context of software components the problem that a component A performs a callback on a component B while B is in an unexpected inconsistent state is known as the *component re-entrance problem*. The definition of contracts for components which are usually developed completely independently of each other becomes more difficult in this case. The component re-entrance problem is described in detail in [Szy98] and some recommendations for the usage of pre- and postconditions for situations with callbacks are provided. A more formal treatment of this problem can be found in [MSL99]. As we shall see, protocol state machines (PSMs) with pre- and postconditions can be used as a technique for modelling reentrance on a component level.

The next subsection demonstrates this using an example which we will use throughout the paper.

2.1. An example of callbacks with UML

The following example is used in different versions to illustrate several problems with callbacks in UML. It is a variation of the Observer pattern [Gra98], [GHJV95] and models parts of the UML tool ArgoUML [Arg]. [3]

The class diagram in Figure 2 specifies that there are two different classes extending the abstract class Observer: ClassDiagramEditor and NavPane, both components of the tool's GUI as shown in Figure 3. They present a graphic view on the Model, which is the UML model created by the user. The Model is the subject under observation and contains model elements such as classes, associations and generalization relations which can be added, changed and removed by particular methods. In this example we concentrate on the removal of model elements and only consider methods which are relevant in this context.

The Observers possess methods which enforce an update when the Model has been changed. There are separate update-methods for the deletion of each kind of model element that is considered here. All of these methods take the element that has been removed from the Model as parameter. That means the Observers get some information about what changes have occurred and do not have to process the complete Model. [4]

In addition to these update-methods ClassDiagramEditor also provides methods for requesting changes

[3] The example has been simplified and adapted to be more suitable for our purpose but the basic concepts are very close to the actual implementation of ArgoUML.

[4] This is sometimes referred to as the *push*-approach, where some information is pushed to the Observers, in contrast to the *pull*-approach where the Observers have to query the Model to get the full state.

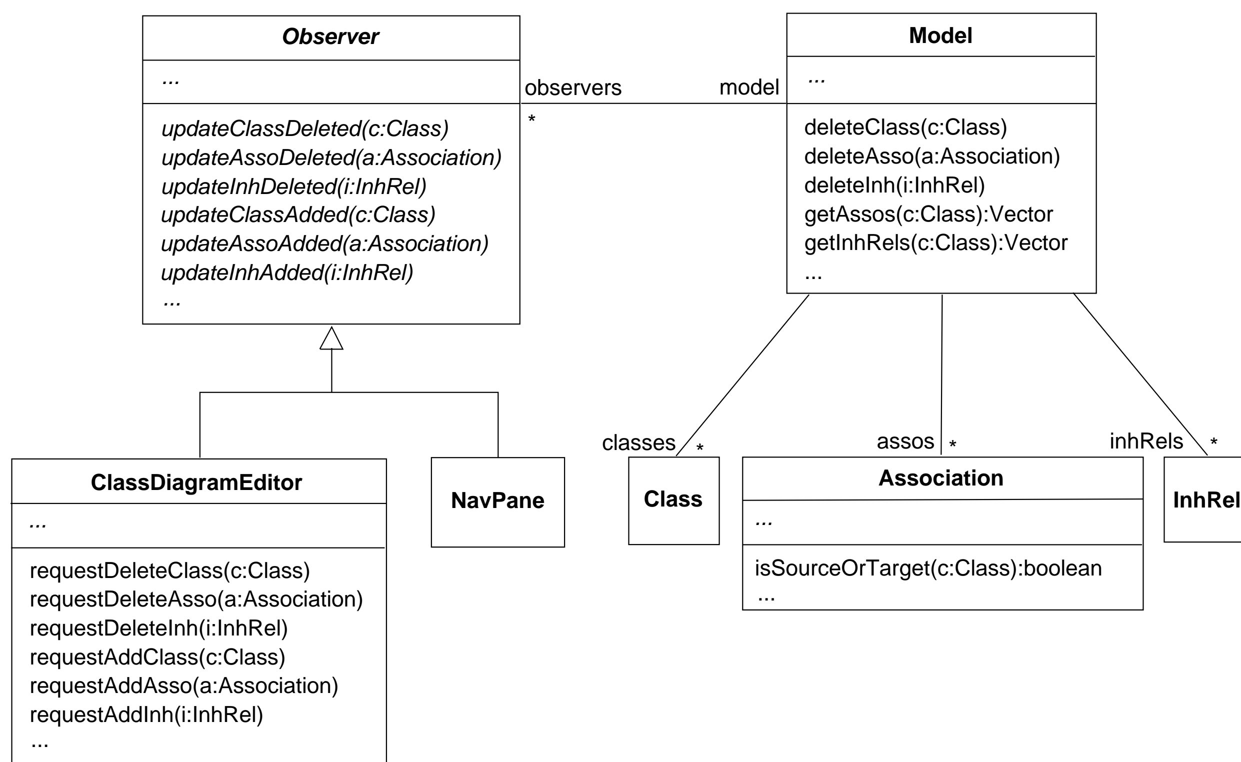

Fig. 2. Callback example – class diagram

of the Model. For example the method requestDeleteClass is invoked if there is a class selected in the editor and the user asks for its deletion via clicking on a menu item.

The interaction between a Model m and two Observers cde and np in response to this request by the user is shown in Figure 4: cde invokes deleteClass on m which then calls a method on itself that yields all associations in which the class that is to be removed participates. These associations are deleted one by one in a loop and in each iteration the Observers are informed about these changes. This procedure ensures that the UML model remains valid and does not contain any "dangling associations".

The ClassDiagramEditor cde which has invoked deleteClass is still waiting for the completion of this method when it receives notifications about deleted associations from m, i.e. m performs a callback on cde. A similar interaction takes place for the removal of all relevant generalization relations. Finally the Observers receive an update that the requested deletion of a class has been completed. For Observer cde this final update is again a callback interaction.

An interaction like that can be implemented in Java without problems. The parts of a possible implementation[5] for Model which are relevant for the callback are shown in Figure 5. The implementation of method deleteInh has been omitted because it is very similar to deleteAsso. If a call of model.deleteClass is part of method requestClassDeleted and the ClassDiagramEditor cde is registered as an Observer of m, a callback as shown in the sequence diagram occurs.

The internal behaviour of objects of classes Model and ClassDiagramEditor can be modelled by UML state machines. We first consider the state machine for Model as shown in Figure 6.

A Model object just has one default state with a loop arrow that shows which methods can be called and what effect they have. Normally a designer would not draw such a state machine because it does not contain any interesting state changes. However, it is certainly a valid UML state machine and we use it here

[5] For simplicity the Java event mechanisms which are essential in the real implementation of ArgoUML are not used here.

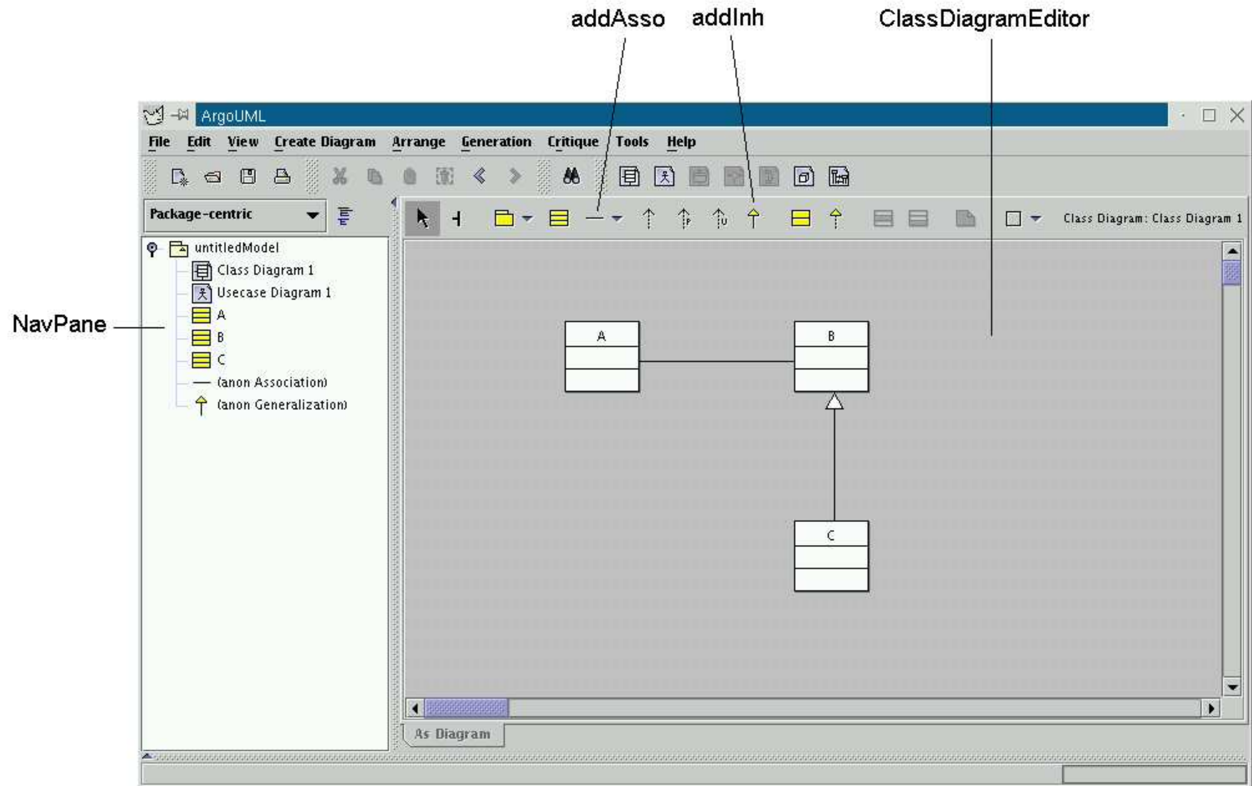

Fig. 3. Parts of the ArgoUML GUI

to demonstrate that the UML state machine semantics generally does not make sense for callbacks, even in such simple cases.

When a call of deleteClass arrives at a Model object m, the loop transition with the corresponding label is triggered. That leads to a call of getAssos on the same object m which has not yet finished the execution of the first transition. The UML run-to-completion semantics prescribes that m can only process the call of getAssos after the loop transition triggered by deleteClass has been completed, which will never happen. Notice that this problem occurs independently of whether the callback method changes the state of an object or not.

For a more complicated example consider the state machine for ClassDiagramEditor, given in Figure 7. For readability, an arrow may represent more than one transition with the same source and target states. The different transition labels are separated by commas. For instance the arrow from state “addInh disabled” to itself represents three transitions. We assume that all methods of ClassDiagramEditor can be invoked on one of its objects when the object is in state “all buttons enabled”, but only show those transition labels here that are of interest for the example.

Depending on the associated Model some of the editor’s buttons may be disabled[6]. This is useful to prevent the user from drawing a UML class diagram that is invalid. The editor is initially in a state where the buttons for adding associations and generalization relations to the class diagram are disabled. These buttons, which are called “addAsso” and “addInh” in this example, are shown and labelled in Figure 3. The user first has to create a class before s/he can define any relationships. Similarly drawing generalization relationships is only possible after the user has added two classes to the diagram. This prevents the creation of a model containing one class with circular generalization, which is forbidden in UML.

[6] This behaviour is not part of ArgoUML but fits well with its aim of guiding the user in the production of valid UML models and serves as an example of callbacks that change an object’s state.

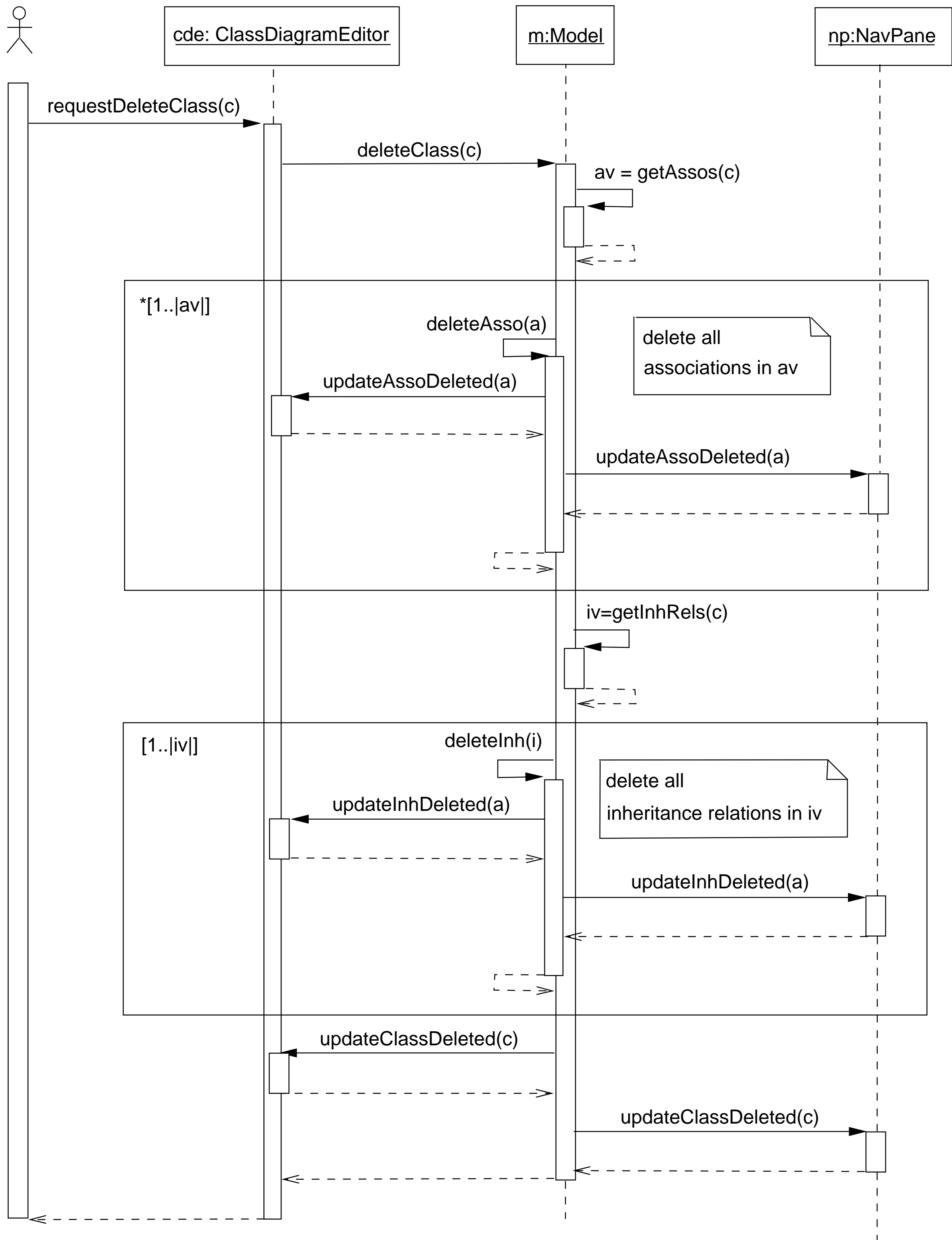

Fig. 4. Callback example – sequence diagram

```
public void deleteClass(Class c) {
        Vector av=getAssos(c);
        for (int j=0; j<av.size(); j++) {
            Association a=(Association)av.get(j);
            deleteAsso(a);
        }
        Vector iv=getInhRels(c);
        for (int j=0; j<iv.size(); j++) {
            InhRel i=(InhRel)iv.get(j);
            deleteInh(i);
        }
        // remove c from classes
        classes.removeElement(c);
        // update observers
        for (int j=0; j<observers.size(); j++) {
            ((Observer)observers.get(j)).updateClassDeleted(c);
        }
    }

    public void deleteAsso(Association a) {
        // remove a from assos
        assos.removeElement(a);
        // update observers
        for (int j=0; j<observers.size(); j++) {
            ((Observer)observers.get(j)).updateAssoDeleted(a);
        }
    }
```

Fig. 5. Parts of a Java implementation for class Model

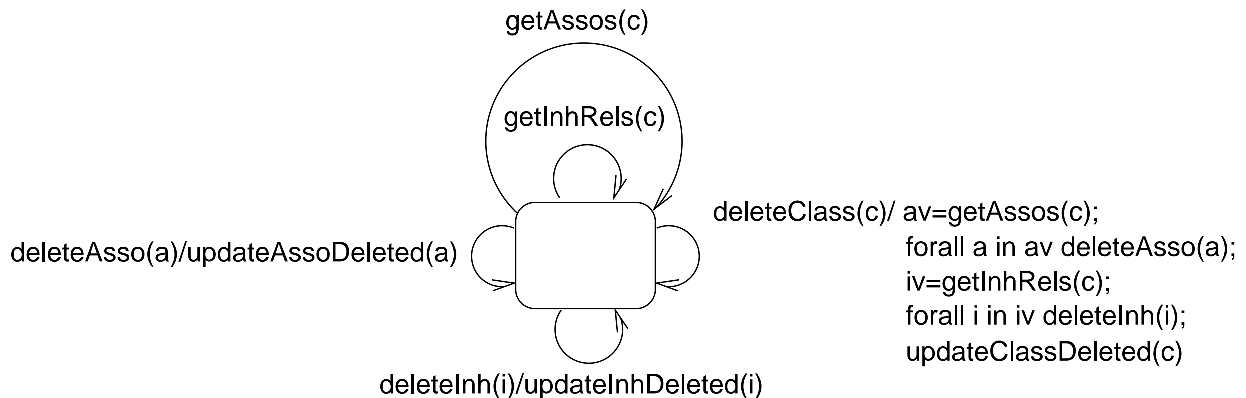

Fig. 6. State machine for Model

If a ClassDiagramEditor is in state "all buttons enabled" arbitrarily many classes can be added without changing the state. In this case a call of updateClassDeleted can either leave all buttons enabled or result in a state change to "addInh disabled", i.e. our state machine is non-deterministic at this point.

Now consider a ClassDiagramEditor cde in state "all buttons enabled" which observes Model m. When requestDeleteClass is called, the corresponding loop transition in the state machine for cde is triggered. During the transition cde calls deleteClass on the associated Model m. From the sequence diagram (and from the state machine for Model) we see that m calls back and invokes updateClassDeleted on cde. Note that this callback always takes place while invocations of updateAssoDeleted and updateInhDeleted only occur if the

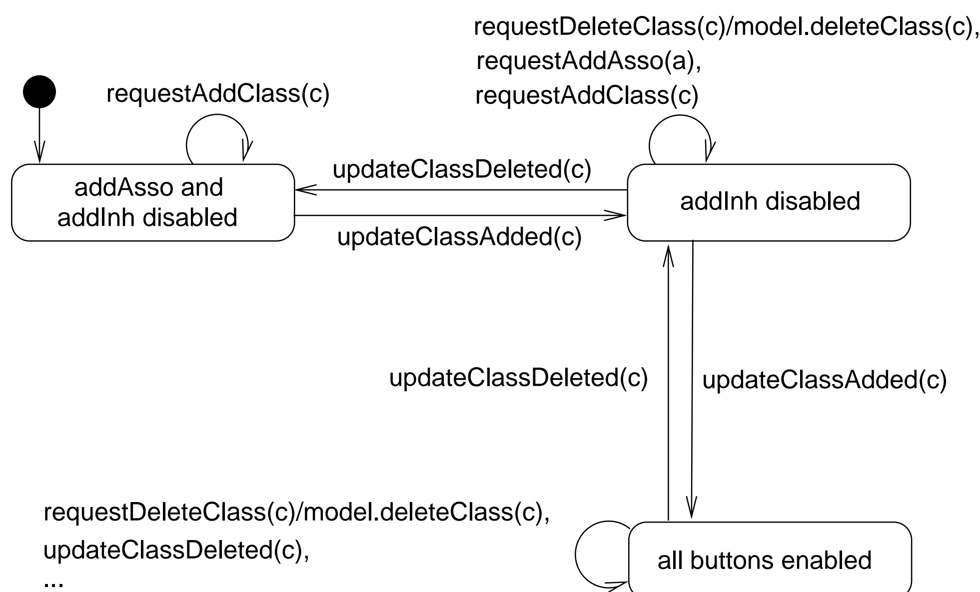

Fig. 7. State machine for ClassDiagramEditor

class that is deleted participates in any relations of these kinds. According to the UML semantics cde cannot react to the call of updatedClassDeleted because it is not in a stable state when the call arrives. This is deeply counterintuitive. The state diagrams shown are intuitively correctly implemented by the Java code shown; the Java code works correctly, without deadlock problems or anything of the kind; and yet, the collection of UML state diagrams has emergent behaviour which fails to capture the correct behaviour of the Java code. A designer working with the UML state diagrams in a tool which enforced the UML semantics might be told that there was a fatal problem with the design, when in fact there is not. The semantics of the collection of UML state diagrams is inconsistent with the semantics of the collection of Java code, even though individually the state diagrams are correctly implemented by the Java code fragments. We do not believe that it is possible to argue that this situation is acceptable. It defeats the purpose of modelling, which is to allow the detection of genuine problems with a design prior to implementation. The UML semantics is broken.

3. Two kinds of state diagrams

We suggest handling the problem of callbacks (including recursion) by using two different kinds of state diagrams, one to model the overall effect of a method on the state of an object and the other (when the modeller considers it appropriate) to model the execution of actions of which this method consists. Thus we resolve the issue of whether state diagrams are loose specifications or executable machines by providing both, for use in clearly defined different contexts.

3.1. Protocol state machines (PSMs)

In UML1.5 [OMG03] (2-165) PSMs are introduced as a state diagram variant, defined in the context of a classifier. We keep UML's terminology (PSM), but in our opinion these diagrams are best thought of as loose *specifications*, not as executable *machines*. They specify the permissible sequences of method calls on an object (the protocol), but not how the object will react to each method call. Their transitions (protocol transitions) are *allowed*, but not expected, to have action expressions. Here we only consider PSMs for classes and in order to enforce the separation between the diagram types, we follow UML2.0, in which actions at protocol transitions are explicitly forbidden. The definition given below is a formalisation of a simplification

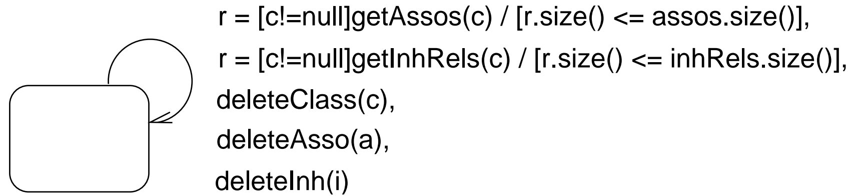

Fig. 8. PSM for Model

of PSMs as presented in [OMG04]. In its simplest form a PSM is a state diagram in standard UML notation whose transitions are triggered by call events and do not have actions: e.g., removing all actions from Figure 7 yields a PSM for ClassDiagramEditor.

In the context of a PSM the guards which can be attached to transitions may represent a part of the method's precondition. (Alternatively, they may demonstrate that state resulting from a method call depends, for example, on the value of the arguments to the method.) Normally the guard will express a restriction on the parameter of the method. Requirements concerning the state of the object, which can be regarded as another part of the precondition, are captured by the protocol states in the PSM.

Similarly we also allow part of an operation's postcondition to be attached to a protocol transition. Such a condition will usually be a restriction on the result of a method. Using an extension of the standard notation transition labels are of the form $r = [pre]m/[post]$, where r is the result of m, m is the method which the transition refers to, *pre* is the condition that has to hold when the call of m occurs for this transition to fire, and *post* is a condition that has to be valid after m has been executed. We refer to *pre* and *post* as the pre- and postcondition of a protocol transition. Notice that the actual execution of m is not modelled by a PSM but by a another diagram type which is introduced later. Figure 8 shows a PSM for class Model which contains protocol transitions with pre- and postconditions.

Protocol states and conditions on the protocol transitions of a PSM can be combined to one constraint by logic connectives, equivalent to writing pre- and postconditions of the corresponding methods in the traditional way (see [OMG04] for details). In practice it might sometimes be useful to factor out pre- and postconditions of protocol transitions, especially those which are common for all protocol transitions labelled by the same method or which are not of particular interest in the context of the PSM.

The designer would develop a PSM only for classes that s/he considers to have interesting state change behaviour, as in current practice. Instead of recording actions on the transitions, s/he will choose when it is worthwhile to record how reactions to events are implemented using another diagram type, as follows.

3.2. Method state machines (MSMs)

Both UML1.5 [OMG03] and the forthcoming UML2.0 [OMG04] allow the definition of a state diagram in the context of an operation, but do not provide detail about the particular features and behaviour of this kind of state diagram. We propose MSMs which are a simplified variant of *sequential class machines* as presented in [GLS+01], which in turn are a variant of *recursive state machines* as introduced in [AEY01]; they allow recursion.

Figure 9 shows an MSM for getAsso, represented by a box with rounded edges and labelled by the class which owns the method, the method name, and its parameters. The MSM consists of an *entry state* E1, an *object creation box*, four *invocation boxes*, two *internal states* S1 and S2, and a *return state* R1.

Much of the notation is standard UML, but with additions as follows. The entry state contains variable declarations and the return state includes a return expression which is compatible with the method's return type. An MSM can have several return states with different return expressions. Object creation boxes have a double borderline and contain the string <<create>> and a class name. Each box has an entry and exit point, represented as shown in Figure 9. The exit point is labelled by a *return variable* which must be declared in the MSM's entry state and whose type must be the class specified in the box. In our example the object creation box contains class name Vector and its exit point is labelled by r, which is of type Vector. Invocation boxes are very similar to object creation boxes but include a method call. The type of the return variable attached to an invocation box's exit point corresponds to the return type of the method given in the box. States and boxes of the MSM are connected by transitions which are labelled by guards and actions, but not

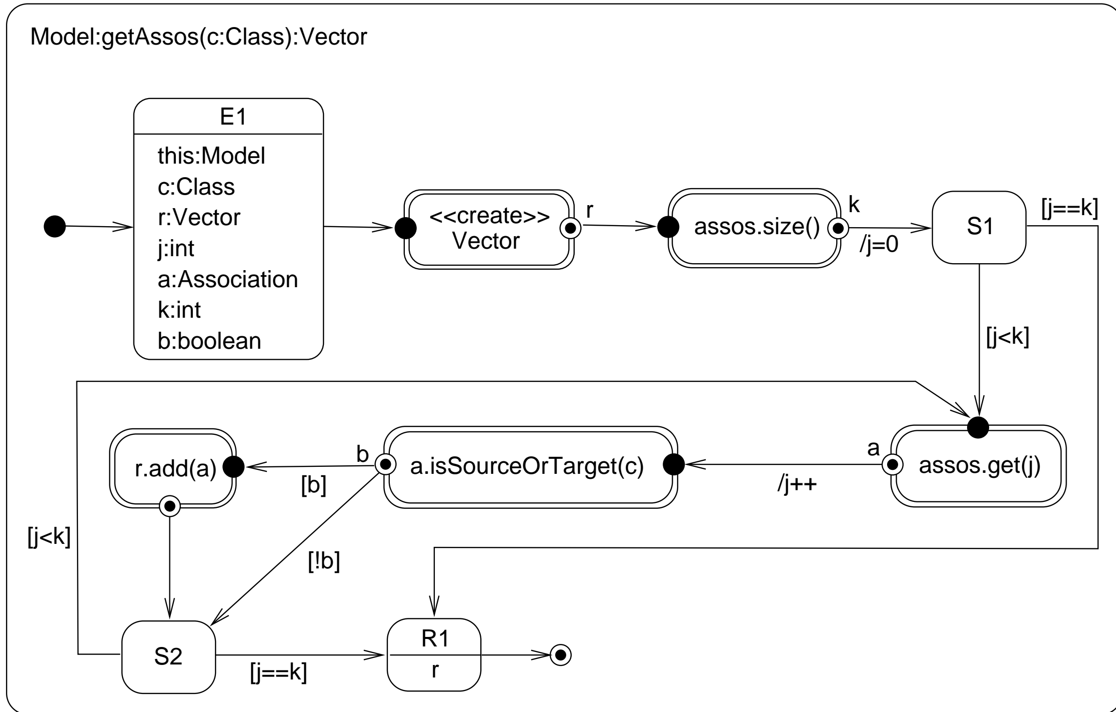

Fig. 9. Method state machine for getAssos

by events. The actions in an MSM can only manipulate local variables or the object on which the method represented by the MSM was called. All method calls have to be included in invocation boxes (as opposed, for example, to being included in actions on transitions, or in any return expression). The MSM for getAssos will be considered in more detail in Section 5.

The basic idea for the execution of MSMs is that they can invoke each other via invocation boxes. When an invocation box is reached during the execution of an MSM, the MSM for the method in this box is invoked. The first MSM only continues its execution after the execution of the second MSM is finished. Figure 10 shows an MSM fragment for each of the three methods that are relevant for our callback example. The execution order is indicated by bold arrows. We will discuss the execution of MSMs more formally in Section 5.1.

In terms of the UML metamodel implicit completion events are relevant for MSMs. They cause normal transitions (as commonly used in UML activity diagrams) and also permit the MSM to move on from a method invocation box when the execution of the MSM for the call in the box is finished. Since an MSM models how an activity is performed, it is bound to have similarities with an activity diagram. We add a precise semantics for MSMs in Section 5, especially, semantics for invocation of methods represented by other diagrams, which is not defined in UML activity diagrams. In the next section we will discuss how PSMs and MSMs sit inside the UML Metamodel.

4. Embedding into the UML Metamodel

Figure 11 shows an extension of the UML1.5 metamodel by protocol state machines.[7] Attributes and operations are only shown where they are of importance for our work. Three new metaclasses have been defined: ProtocolStateMachine as subclass of StateMachine, ProtocolTransition as subclass of Transition, and PostCondition. Between ProtocolTransition and PostCondition an association has been added which allows a

[7] Note that our extension has the effect of bringing the metamodel closer to that of UML2.0: see discussion later.

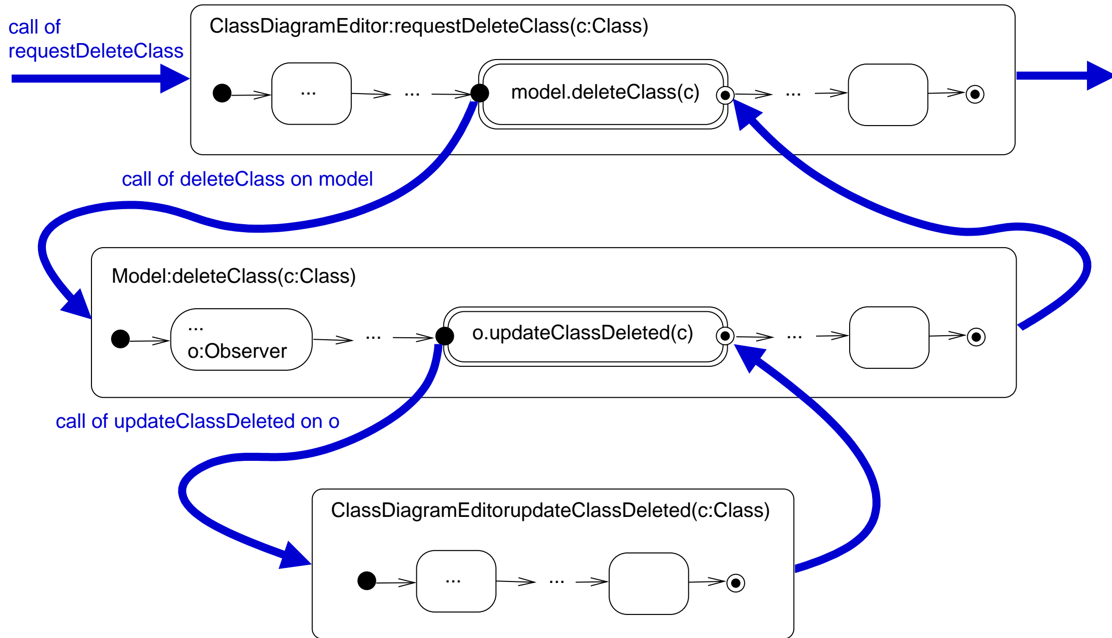

Fig. 10. Execution order of MSMs for callback example

postcondition to be attached to a ProtocolTransition. Notice that we have not introduced a similar construction for preconditions. Instead we reuse the metaclass Guard which is part of the original metamodel and can be associated with any transition for defining a precondition. However, a guard in the context of the original metamodel prevents transitions from being fired under certain conditions. This does not apply to preconditions in PSMs, which cannot be executed and where therefore the idea of transitions being triggered and fired does not make sense. A precondition is just a part of the method's specification.

In Section 4.1 constraints for protocol state machines are given as well-formedness rules. These rules capture for instance which vertices are valid in a PSM. This is necessary because we did not add a class ProtocolVertex to the metamodel and only a subset of StateVertex instances is valid in PSMs. Well-formedness rules have been used here because they are simpler to define than a new metaclass ProtocolVertex in the context of the current UML model which contains a complicated generalization hierarchy below StateVertex.

The well-formedness rules also specify other constraints on PSMs such as, for example, that each ProtocolTransition has a trigger of type CallEvent. Since each call event is connected to an operation, an association from ProtocolTransition to Operation can be derived as shown in Figure 11. Note that *operations* and *methods* are distinguished in UML in order to allow for inheritance: classes in an inheritance hierarchy may all have the same operation, inherited from a base class, which they implement using different methods. We have taken this into account for the embedding into the metamodel, but do not consider inheritance in our examples and formalisation.

For method state machines the UML Metamodel has been extended in similar fashion as shown in Figure 12. Again new subclasses of StateMachine and Transition have been introduced. Additionally the original metamodel is extended by classes BoxEntryState, BoxExitState and ReturnState which are all subclasses of State, and a new class InvocationBox which is a special StateVertex.

As mentioned in Section 3.2 an InvocationBox has a BoxEntryState and BoxExitState attached to it which is represented by the associations between these classes in Figure 12. Furthermore the extended metamodel contains associations modelling that a set of local variables is defined for an MSM, an MSM always refers to an Operation, a BoxExitState may have a variable attached to it, and a ReturnState may contain a return expression.

As for PSMs well-formedness rules in Section 4.1 specify which vertices are valid in an MSM and which other conditions have to be fulfilled by a well-formed MSM.

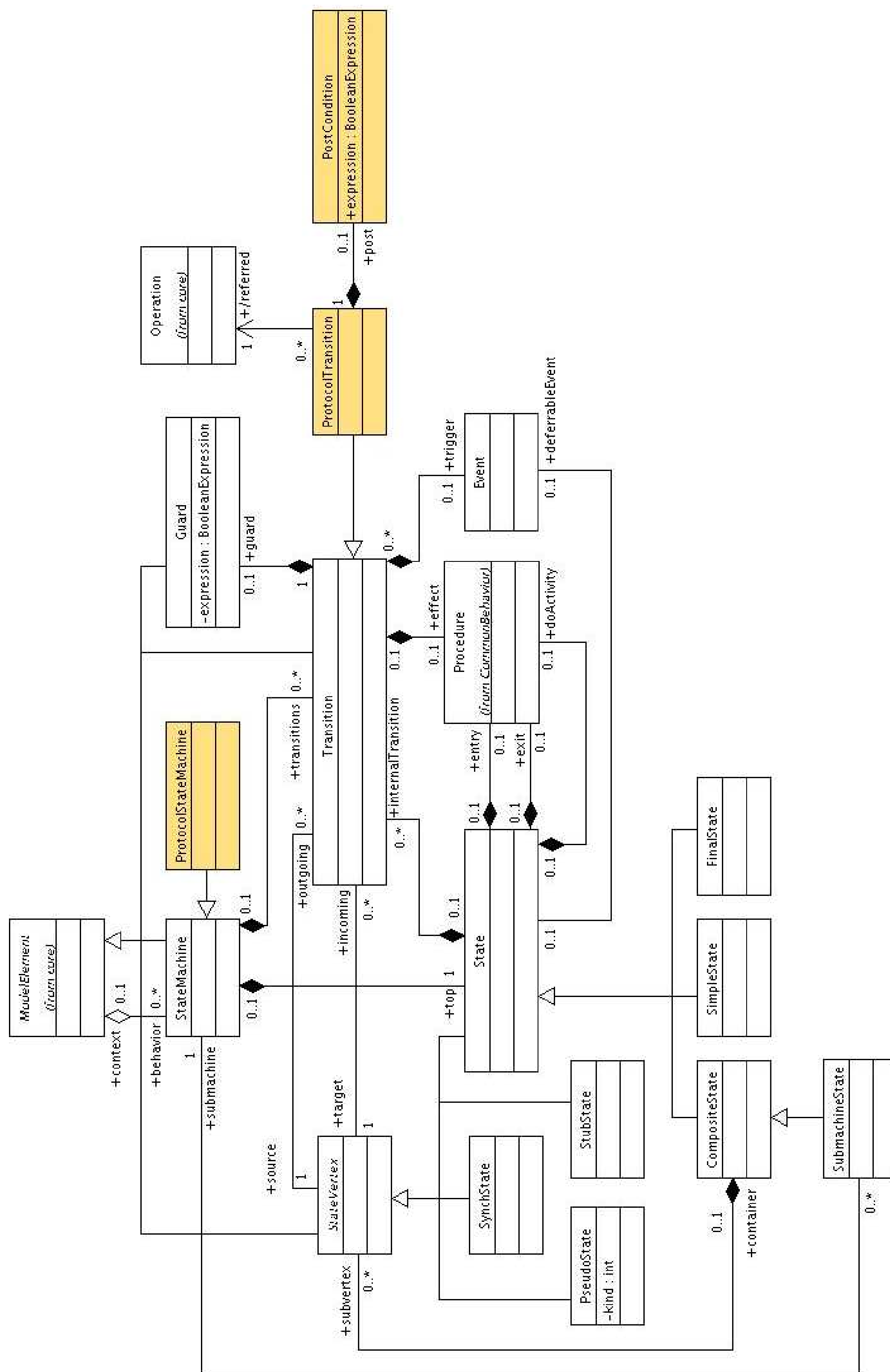

Fig. 11. Embedding of protocol state machines into the UML1.5 Metamodel

MSMs and PSMs are connected via their relations to Operation in the metamodel. Protocol transitions in a PSM may refer to an operation which is realised by an MSM.

4.1. Well-formedness rules

The well-formedness rules for MSMs and PSMs together with additional operations needed for their specification are given in the Object Constraint Language (OCL) [OMG03](Chapter 6), [WK99]. Some of these rules were taken from UML2.0[OMG04] and the earlier U2P proposal [Par03] and slightly modified. There are some points that have to be noticed with respect to the usage of OCL in this section:

- the difference between operators `oclIsKindOf` and `oclIsTypeOf` is that the former refers to the direct type and indirect supertypes of an object, while the latter refers to the direct type. That means for an object `o` of type `C` with `C` a subtype of `D`, `o.oclIsTypeOf(D)` evaluates to `false` but for `o.oclIsKindOf(D)` the result is `true`.
- some rules do not impose further restrictions on PSMs and MSMs but were added because they express important properties of model elements which are guaranteed by other rules.
- there is no special notation for upcasting. OclAsType is used for casting an object down to a specific subtype.[8]

ProtocolTransition

1. A ProtocolTransition always belongs to a ProtocolStateMachine.
   ```
   self.stateMachine.oclIsTypeOf(ProtocolStateMachine)
   ```
2. A ProtocolTransition never has a procedure associated as effect.
   ```
   self.effect → isEmpty()
   ```
3. The trigger associated with a ProtocolTransition is never empty and is a call event.
   ```
   self.trigger → notEmpty() and
   self.trigger.oclIsTypeOf(CallEvent)
   ```
4. A ProtocolTransition refers to an operation, reachable via its trigger.
   ```
   self.referred =
       self.trigger.oclAsType(CallEvent).operation
   ```
5. The referred operation of a ProtocolTransition is owned by the Classifier that is the context of the ProtocolStateMachine the ProtocolTransition belongs to.
   ```
   self.referred.owner=
       self.stateMachine.context.oclAsType(Classifier)
   ```
6. The source and target of a ProtocolTransition are PSM vertices.
   ```
   self.source.isValidPSMVertex() and self.target.isValidPSMVertex()
   ```
   This is guaranteed by rule 3 for ProtocolStateMachine.

ProtocolStateMachine

1. A ProtocolStateMachine must have a classifier context.
   ```
   self.context.oclIsKindOf(Classifier)
   ```
2. All transitions in a ProtocolStateMachine must be ProtocolTransitions.
   ```
   self.transitions
       → forAll(t|t.oclIsTypeOf(ProtocolTransition))
   ```
3. All vertices in a ProtocolStateMachine must be valid PSM vertices.
   ```
   self.top.oclAsType(CompositeState).getAllSubvertices
       → forAll(v|v.isValidPSMVertex())
   ```

[8] Note that some versions of the OCL specification used OclAsType inconsistently for both up- and down-casting.

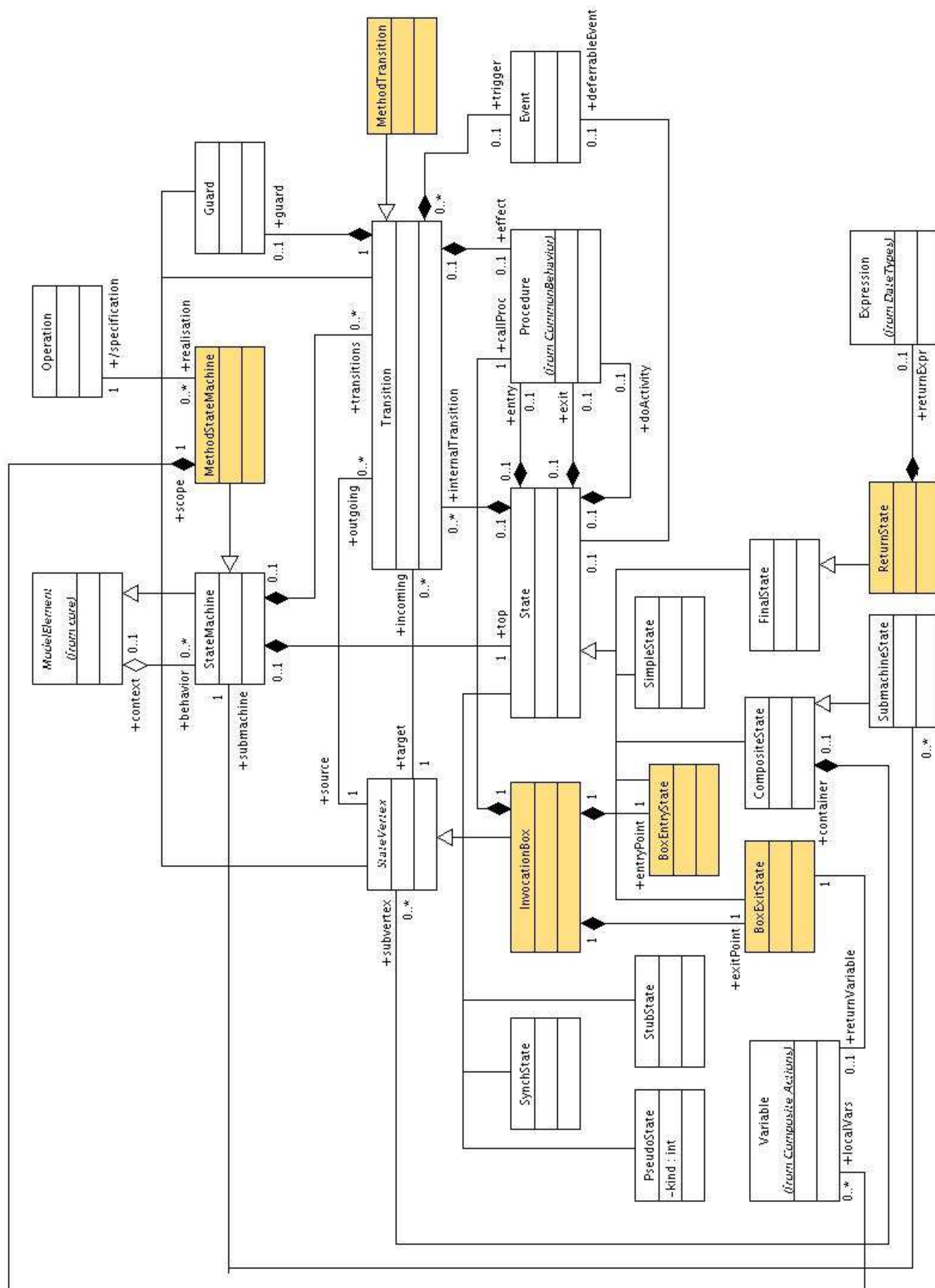

Fig. 12. Embedding of method state machines into the UML1.5 Metamodel

MethodTransition

1. A MethodTransition always belongs to a MethodStateMachine.
   `self.stateMachine.oclIsTypeOf(MethodStateMachine)`
2. If a MethodTransition has an associated effect then the effect does not contain any method calls[9].
3. The trigger associated with a MethodTransition is empty[10].
   `self.trigger → isEmpty()`
4. The source and target of a MethodTransition are MSM vertices.
   `self.source.isValidMSMVertex() and self.target.isValidMSMVertex()`
   This is guaranteed by rule 4 for MethodStateMachine.

MethodStateMachine

1. A MethodStateMachine must have a method context.
   `self.context.oclIsTypeOf(Method)`
2. A MethodStateMachine is specified by an operation, reachable via its context.
```
self.specification=
    self.context.oclAsType(Method).specification
```
3. All transitions in a MethodStateMachine must be MethodTransitions.
```
self.transitions
    → forAll(t|t.oclIsTypeOf(MethodTransition))
```
4. All vertices in a MethodStateMachine must be valid MSM vertices.
```
self.top.oclAsType(CompositeState).getAllSubvertices
    → forAll(v|v.isValidMSMVertex())
```

ReturnState

1. The return expression associated with a return state does not contain any method calls.[11]

InvocationBox

1. The call procedure associated with an invocation box is a single action which is an invocation action.
```
self.callProc.action.oclIsKindOf(PrimitiveAction) and
    (self.callProc.action.oclIsKindOf(ExplicitInvocationAction) or
        self.callProc.action.oclIsKindOf(InvocationAction))
```

BoxExitState

1. The return variable of a box exit state is in the set of local variables defined for the MSM the box exit state belongs to.
```
self.stateMachine.oclAsType(MethodStateMachine).localVars
    → includes(self.returnVariable)
```

Additional operations

1. An operation that collects all vertices contained in a CompositeState recursively.

[9] A more precise definition in OCL is omitted because due to the nature of the UML Action semantics it would be a fairly complicated constraint which does not bring further clarification.
[10] Well-formedness rule 5 for StateMachine in the current UML specification [OMG03](2-152) which applies to state machines describing a behavioural feature is similar to this but tries to specify an exceptional case where transitions emerging from an initial pseudostate may have a trigger and refers only to call events. Unfortunately this rule and rules 5 and 6 for Transition [OMG03](2-154) do not seem to be consistent in the standard.
[11] Since in accordance with UML1.5's Expression metaclass we do not specify the expression language, we cannot express this formally in OCL.

```
context CompositeState::getAllVertices():Set(StateVertex)
   result = self.subvertex → union (
              (self.subvertex → select(oclIsTypeOf(CompositeState)))
                → iterate(
                     c:CompositeState;
                     sv:Set(StateVertex)=Set{}
                     | sv → union(c.getAllSubVertices())
                )
            )
```

2. A StateVertex is valid in a PSM if it is either a valid PSM state or a valid PSM PseudoState.

```
context StateVertex::isValidPSMVertex():Boolean
   result = if self.oclIsKindOf(State) then
               self.oclAsType(State).isValidPSMState()
            else if self.oclIsKindOf(PseudoState) then
               self.oclAsType(PseudoState).isValidPSMPseudoState()
            else false
```

3. A State is valid in a PSM if its entry, exit, do-activity and internal transition are empty.

```
context State::isValidPSMState():Boolean
   result = self.entry → isEmpty() and
            self.exit → isEmpty() and
            self.doActivity → isEmpty() and
            self.internalTransition → isEmpty()
```

4. A PseudoState is valid in a PSM if it is not a history state.

```
context State::isValidPSMPseudoState():Boolean
   result = self.kind <> #deepHistory and
            self.kind <> #shallowHistory
```

5. A StateVertex is valid in an MSM if it is either a valid MSM state or a valid MSM PseudoState.

```
context StateVertex::isValidMSMVertex():Boolean
   result = if self.oclIsKindOf(State) then
               self.oclAsType(State).isValidMSMState()
            else if self.oclIsKindOf(PseudoState) then
               self.oclAsType(PseudoState)isValidMSMPseudoState()
            else false
```

6. A State is valid in an MSM if its entry, exit, do-activity and internal transition are empty. A FinalState is only a valid State if it is of type ReturnState.

```
context State::isValidMSMState():Boolean
   result = self.entry → isEmpty() and
            self.exit → isEmpty() and
            self.doActivity → isEmpty() and
            self.internalTransition → isEmpty() and
            (self.oclIsKindOf(FinalState) implies
               self.oclIsTypeOf(ReturnState))
```

NB: Notationally, we distinguish ReturnStates from other states by attaching the "bull's eye" symbol normally used for FinalStates. Notice that the state and the bull's eye together form the graphical representation (the concrete syntax) for the single abstract syntax element which is a ReturnState. As in Figure 13, we may attach the same bull's eye to several state boxes; this should be seen in the obvious way as concrete syntactic sugar for the diagram variant which would include several bull's eyes, one attached to each state box, the abstract syntax being unaffected.

7. A PseudoState is valid in an MSM if it is not a history state.

```
context State::isValidMSMPseudoState():Boolean
   result = self.kind <> #deepHistory and
            self.kind <> #shallowHistory
```

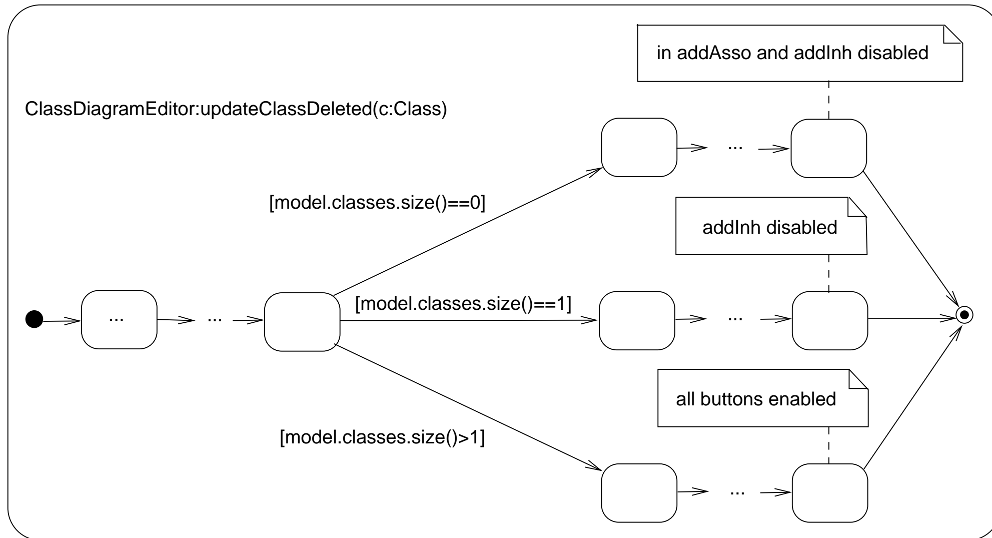

Fig. 13. Extended MSM fragment for updateClassDeleted

5. Formal definitions and consistency

Clearly, the designer will need to satisfy him/herself that the state diagrams, both PSMs and MSMs, contained in a given model are consistent: that is, that it is possible to implement methods according to the MSMs and have the resulting classes act in accordance with the PSMs. In this section we specify what this consistency means.

First we consider some informal examples:

- The MSM for getAssos in Figure 9 is intuitively consistent with the PSM for Model as shown in Figure 8. Assume that the MSM is invoked with parameter c and the precondition c!=null at the corresponding loop transition in the PSM holds. During the execution of the MSM a Vector r is created and filled with all associations which have c as source or target. We assume that isSourceOrTarget can be used to check this property. The loop in which elements are added to r is executed k times where k=assos.size(). That means the size of r can be at most assos.size(), i.e. the postcondition of the protocol transition is fulfilled. The protocol transition has the same protocol state as source and target and thereby reflects that the MSM does not change the protocol state of the Model object it is invoked on.
- If the MSM for getAssos added additional elements to r after looping through assos, the postcondition would not hold in all cases and the consistency between MSM and PSM would be violated.
- Figure 13 shows an extension of the MSM fragment for updateClassDeleted which has been considered before. Without going into details we assume that the following case distinction takes place in the MSM: if the model associated with the ClassDiagramEditor on which updateClassDeleted is invoked contains no classes, then the MSM performs actions such that the ClassDiagramEditor is in state "addAsso and addInh disabled" after the execution of the MSM is finished. In the case that the number of classes in the model is 1, the ClassDiagramEditor is in state "addInh disabled" after updateClassDeleted. Finally, if the model contains more than one class, then the actions of the MSM cause a state change to "all buttons enabled".

  On first sight the behaviour sketched here seems to fit well with the protocol specified by the PSM, but in fact MSM and PSM are not consistent with each other in this case. The problem here is that the execution of the MSM depends on the size of classes which belongs to another object. So far we have not formally specified any connection between the state of the ClassDiagramEditor and the number of classes in its associated Model. For instance, a call of updateClassDeleted could arrive at a ClassDiagramEditor in state "all buttons enabled" when the number of classes in the associated model is zero. The MSM will change the state of the ClassDiagramEditor to "addAsso and addInh disabled" in this case. Since there is no corresponding protocol transition from "all buttons enabled" leading to this state in the PSM,

consistency is violated. Similar problems arise for other combinations of ClassDiagramEditor state and number of classes in the model. The informal description of our example in Section 2.1 implies that none of these erroneous combinations should occur in the system.
A simple solution to this problem is, for example, to add a loop transition labelled by updateClassDeleted to each state of the PSM which does not have such a transition yet. The MSM has to be changed such that it includes a test whether a combination of ClassDiagramEditor state and number of classes in its model is valid. If it is illegal, an error message is printed and the MSM does not change the object's protocol state. If the combination is valid, the execution should proceed as shown in Figure 13. A more realistic alternative would be to throw exceptions in the MSM and to define an explicit error state in the PSM.
The dependency between ClassDiagramEditor and Model could also be made explicit by an invariant, for instance in OCL. However, we do not consider the effect of invariants on PSMs and MSMs in this paper, but may do so in future work.

In order to formalise these intuitions of consistency we introduce formal definitions of both kinds of state diagrams. For purposes of exposition we do not cover all of the UML notation and use simplified forms of the diagrams. For instance, we do not explicitly consider hierarchical state machines, because each hierarchical state machine can be "flattened", i.e. transformed into an equivalent state machine without nested states. We believe that most of the features which are missing and do not have a semantic equivalent in our simplified version of state machines could be added without serious problems. Our definitions of MSMs and their execution are adapted from definitions in [GLS+01] and [AEY01].

We assume that there is a class diagram which defines a set of classes C. Each $c \in C$ is associated with a finite set $A_c = \{a_1 : T_1, \dots, a_n : T_n\}$ of typed attributes and a finite set M_c of methods, where each method $m \in M_c$ has a type $T_{cm} = cT_{cm} \times rT_{cm}$ defining the call and return type of the method[12]. We allow a special empty type *void* as call and return type for methods. Constructors of a class c are typed as ordinary methods with return type c. We only consider a default constructor which does not take any parameters.

Remember that we do not consider inheritance in our formalisation, since this raises many interesting questions about the appropriate inheritance of behaviour, and so the distinction between operation and method is unimportant here. One possibility to explore would be that the protocol state machine would be written in terms of operations, and that the PSM defined for a base class would be inherited by subclasses. Then when a subclass provided its own method implementing an inherited operation, the designer could draw a new MSM for that method. This would open the door to considerations of behavioural subtyping: we could ask to what extent the MSMs for different methods implementing an operation were compatible.

A PSM is unsurprisingly simply a labelled transition system with pre- and postconditions:

Definition 5.1. A protocol state machine (PSM) for a class c consists of

- a set S_c of states
- a labelled transition relation $\gamma \subseteq S_c \times L \times S_c$ where each label $l \in L$ is of the form $r = [pre]m(x)/[post]$ where
  - r is a return parameter of type rT_{cm},
  - $m \in M_c$ is a method name,
  - x is a formal parameter of type cT_{cm},
  - *pre* is a Boolean expression over $A_c \cup \{x\}$ specifying the condition under which the transition may be taken,
  - *post* is a Boolean expression over $A_c \cup \{r, x\}$ specifying a condition which must hold after the execution of m has been completed.

We do not prescribe the expression language used for *pre* and *post* but we assume that an expression can be evaluated to true or false given values for $A_c \cup \{x\}$ and $A_c \cup \{r, x\}$, respectively.

We will now define MSMs more formally. For a set $X = \{x_1 : T_1, \dots, x_n : T_n\}$ of typed variables, a *variable environment* σ over X is a function $[x_1 \mapsto a_1, \dots, x_n \mapsto a_n]$ where $a_i \in T_i \cup \bot_{T_i}$ for all i. The set of all variable environments over X is denoted by Σ_X. Attributes and attribute environments are treated in a

[12] For simplicity we allow only one parameter and return value

similar way. For the creation of new objects we assume that there exists a default attribute environment τ_c over A_c for each class c.

Moreover let $A = \bigcup_{c \in C} A_c$ be the set of all attributes and O the set of all object identifiers. An *object environment* is defined as a partial function $\omega : C \rightarrow (O \rightarrow \Sigma_A)$ and the set of all object environments is denoted by Ω.

We do not prescribe an action language: we only specify that, given an object environment ω and variable environment σ over X, an action is syntactically an expression over X, suitably extended with attribute selectors, for which an evaluation function $[\![\ ]\!]_{\sigma\omega}$ exists. We will later assume that the same evaluation function can be used to evaluate the pre- and postconditions used in PSMs. An action may not involve the invocation of methods or the creation or deletion of objects. Semantically an action α is a partial function $\alpha : (\Sigma_X \times \Omega) \rightarrow (\Sigma_X \times \Omega)$ expressing the effect the action has on the variable environment and object environment.

Definition 5.2. A method state machine (MSM) for a method $m \in M_c$ consists of

- a set of local variables $X_{cm} = \{x_1 : T_1, \ldots, x_n : T_n\}$, including those mentioned below
- a set B_{cm} of boxes as defined below
- a set of states S_{cm} partitioned into
  - a set I_{cm} of internal states
  - an entry state e_{cm} with formal parameters $x : cT_{cm}$ and $\mathsf{this} : c$ in X_{cm}
  - A set of return states R_{cm} where each state $r \in R_{cm}$ has attached a return expression re over X_{cm} of type rT_{cm}
  - a set of box entry points Entry_{cm} and a set of box exit points Exit_{cm}
- a transition relation $\delta_{cm} \subseteq F \times Act \times T$ where $F = \{e_{cm}\} \cup I_{cm} \cup \mathrm{Exit}_{cm}$, $T = R_{cm} \cup I_{cm} \cup \mathrm{Entry}_{cm}$ and Act is a set of actions $\alpha : (\Sigma_{X_{cm}} \times \Omega) \rightarrow (\Sigma_{X_{cm}} \times \Omega)$

Notice the "incompleteness" of the transition relation of an individual MSM: if the MSM reaches a box entry point, it cannot go further based on the definition of this MSM alone. This makes sense because we cannot know what the effect of the call on the environments should be. Later we will show how several MSMs interact to "complete" the transition relation.

Definition 5.3. A box $b \in B_{cm}$ is not itself considered to be a state in the MSM: instead it has two associated states:

- an entry point $c_b \in \mathrm{Entry}_{cm}$
- an exit point $r_b \in \mathrm{Exit}_{cm}$.

A *return variable* y from X_{cm} is defined to hold the value returned from a box. There are two different types of boxes which are defined as follows.

Definition 5.4. An object creation box $cb \in B_{cm}$ specifies a class identifier $d \in C$ determining the class of which a new object is to be created. The return variable of cb is of type d.

Definition 5.5. A method invocation box $mb \in B_{cm}$ specifies

- an object expression oe, determining the target object
- a class identifier $d \in C$ determining the class[13] of oe
- a method identifier $l \in M_d$
- an argument expression $ae : cT_{dl}$

The return variable for box mb is of type rT_{dl}.

Notice that any MSM is by definition well-typed, and that all method invocations occur in boxes.

[13] As mentioned, we do not consider inheritance in this work, so polymorphism is not allowed: the class of the target object must be given statically.

5.1. Execution of MSMs

Suppose that we have a *closed set* $\mathcal{MSM}$ of MSMs: that is, each method invoked in an invocation box of an MSM in $\mathcal{MSM}$ is itself defined by an MSM in $\mathcal{MSM}$. We can then define the execution of MSMs in terms of a global state machine.

Let the sets of states and of boxes for each MSM be pairwise disjoint and let N be the set of all states, X the set of all variables, and B the set of all boxes. Before we specify a global state machine, we give definitions of a call stack and a global environment.

Definition 5.6 (Call stack). A call stack $cs \in B^*N$ specifies the current position in each active MSM. It is a stack $b_1 : \ldots : b_k : n$ of boxes b_i and a state n on top. It must satisfy a coherence condition as follows. Suppose that box b_j contains method identifier m_{b_j} and class identifier c_{b_j}. Then box b_{j+1} if $j < k$ (respectively the state n if $j = k$), must belong to the MSM for method m_{b_j} in class c_{b_j} This MSM must exist, by the assumption that we have a closed set of MSMs.

Definition 5.7 (Global environment). Given a call stack $cs = b_1 : \ldots : b_k : n$, a global environment $ge = \sigma_0 : \ldots : \sigma_k \in \Sigma_X^*$ associated with cs is a stack of variable environments. It must satisfy a coherence condition as follows. For each $j \leq k$, σ_j is the local variable environment of the MSM containing box b_{j+1} if $j < k$, or of the MSM containing n otherwise.

Definition 5.8 (Global state machine). A state of a global state machine (GSM) consists of a call stack $cs \in B^*N$, a global environment ge associated with cs, and an object environment ω.

There are four kinds of transitions: as in a pushdown system, the applicable transitions are always determined by the state at the head of the stack. Suppose $cs = b_1 : \ldots : b_k : n$ where n is a state in $M \in \mathcal{MSM}$, and let the global environment be $ge = \sigma_0 : \ldots : \sigma_k$ and the object environment be ω.

1. If n is an entry state, an internal state or a box exit state, the only possible transitions are *internal transitions* which are induced by transitions of M. Formally, suppose that $n \xrightarrow{\alpha} n'$ is a transition in M. Then

$$(b_1 : \ldots : b_k : n, \sigma_0 : \ldots : \sigma_k, \omega) \to (b_1 : \ldots : b_k : n', \sigma_0 : \ldots : \sigma'_k, \omega')$$

is a transition in the global state machine, provided that $\alpha(\sigma_k, \omega) = (\sigma'_k, \omega')$. Note that if (σ_k, ω) is not in the domain of the partial function α, there is no transition.

2. If n is a box entry state for an object creation box which we now call b_{k+1}, the only possible transition is a *creation transition*, creating a new object and adding it to the object environment. If $n = c_{b_{k+1}}$, the entry state for b_{k+1} and $r_{b_{k+1}}$ its exit state, let c be the class specified in b_{k+1}. Then let ω' be the object environment ω updated by binding a fresh object identifier o of class c to the default attribute environment, i.e. $\omega'(c)(o) = \tau_c$, and let σ'_k be the environment σ_k updated by binding the return variable of box b_{k+1} to o. Then

$$\begin{aligned}&(b_1 : \ldots : b_k : c_{b_{k+1}}, \sigma_0 : \ldots : \sigma_k, \omega)\\ &\to (b_1 : \ldots : b_k : r_{b_{k+1}}, \sigma_0 : \ldots : \sigma'_k, \omega')\end{aligned}$$

is a transition of the global state machine.

3. If n is a box entry state for a method invocation box which we now call b_{k+1}, the only possible transition is a *call transition*, pushing a new invocation onto the stack. If $n = c_{b_{k+1}}$, the entry state for b_{k+1}, let the object expression, class, method and argument expression specified in b_{k+1} be oe, c, m and ae respectively. Then let σ_{k+1} be a new variable environment over X_{cm} in which the formal parameter of m and this are bound to $[\![ae]\!]_{\sigma_k \omega}$, $[\![oe]\!]_{\sigma_k \omega}$ respectively. (If either evaluation fails, there is no transition). Let e_{cm} be the entry state of the MSM for method m in class c. Then

$$\begin{aligned}&(b_1 : \ldots : b_k : c_{b_{k+1}}, \sigma_0 : \ldots : \sigma_k, \omega)\\ &\to (b_1 : \ldots : b_k : b_{k+1} : e_{cm}, \sigma_0 : \ldots : \sigma_k : \sigma_{k+1}, \omega)\end{aligned}$$

is a transition of the global state machine.

4. If n is a return state, the only possible transition is a *return transition*, popping the stack. If $n = r \in R_{cm}$, let $r_{b_{k-1}}$ be the exit state for box b_{k-1}, and σ'_{k-1} the environment σ_{k-1} updated by binding the return variable of box b_{k-1} to $[\![re]\!]_{\sigma_k \omega}$, where re is the return expression associated with r. (Again, if re fails to evaluate, there is no transition). Then

$$(b_1 : \ldots : b_k : r, \sigma_0 : \ldots : \sigma_{k-1} : \sigma_k, \omega)$$
$$\rightarrow (b_1 : \ldots : b_{k-1} : r_{b_{k-1}}, \sigma_0 : \ldots : \sigma'_{k-1}, \omega)$$

is a transition of the global state machine.

Note that the only non-determinacy in the global state machine is that arising from non-determinacy inside individual MSMs: if they are deterministic, so is the GSM. Notice also that the behaviour of the GSM respects the stack discipline. We will be most interested in how the GSM implements a particular method call. We write $s \overset{c,m}{\twoheadrightarrow} s'$ when for some class c and method m, $s = (b_1 : \ldots : b_k : e_{cm}, \sigma_0 : \ldots : \sigma_k, \omega)$, $s' = (b_1 : \ldots : b_k : r, \sigma_0 : \ldots : \sigma'_k, \omega')$ for some return state $r \in R_{cm}$ and there is some sequence of GSM transitions $s \rightarrow \ldots s'$, in which no intermediate state whose call stack contains at most k boxes has e_{cm} at the head of the call stack. Without the restriction on intermediate states we might inadvertently "catch" more than one invocation of m from within MSMs that have been activated earlier. Notice that if m is recursive the call stack grows each time m is invoked so that e_{cm} is allowed to appear as head of the call stack in this case.

5.2. Consistency between protocol and method state machines

So far MSMs and PSMs are only connected by method names which are used to label transitions in PSMs and represent the context of an MSM. In this section we define what it means for an MSM to conform to a protocol which is specified by a PSM.

There can be no formal connection unless the designer has specified the precise meanings of the states in the PSM.[14] Accordingly, we assume that along with any PSM P we are given a function h_P which maps an attribute environment to a state in P, which allows the following definition of an abstract state function.

Definition 5.9 (Abstract state function). Suppose we are given a PSM P for class c with a function $h_P : \Sigma_{A_c} \rightarrow S_c$ and a global state machine with a set of states S_g. For a global state $s = (cs, \sigma, \omega)$ in S_g the abstract state function $h^\sharp_P : S_g \rightarrow S_c$ is defined by

$$h^\sharp_P(s) = h_P(\omega(c)(\sigma(\mathsf{this})))$$

Notice that $h^\sharp_P$ is undefined for global states whose local variable this is not bound to an object of class c.

Definition 5.10 (Consistency). Let G be a GSM defined by a closed set $\mathcal{MSM}$ of MSMs, and let P be a PSM for class c. G conforms to P with respect to a given initial global state s if and only if whenever

$$s \rightarrow^* s' \overset{c,m}{\twoheadrightarrow} s''$$

(that is, a method is executed from some global state which is reachable from the initial state) with $s' = (cs', \sigma', \omega')$ and $s'' = (cs'', \sigma'', \omega'')$, we have $h^\sharp_P(s') = p$ and $h^\sharp_P(s'') = p'$ where $p \xrightarrow{r=[pre]m(x)/[post]} p'$ is a transition of P, $[\![pre]\!]_{\sigma'\omega'} = \mathsf{true}$ and $[\![post]\!]_{\sigma''\omega''} = \mathsf{true}$

Note that because the PSM plays no role in the execution of the global state machine, but acts as an independent specification of what it should achieve, it suffices to specify consistency with one PSM at a time. Note also that we are *not* requiring that every transition in the PSM has some counterpart in the GSM. This is deliberate: the PSM for a reusable class will specify all the capabilities of the class, not all of which may be used in a particular system (GSM).

Since we are requiring that every method execution in the GSM is reflected in some PSM transition, the reader can see that we are (as usual) following UML1.5 in expecting that any operation which may be invoked when the object is in a given state is shown on a transition originating in that state in the PSM. If the operation is permitted, but will not cause a state change in the PSM, it is still shown, on a transition from that state to itself. In UML2.0 the rules are slightly different: any operation not causing a state change is permitted in any state, without needing to be shown on the PSM. The best way to handle this seems to be to regard the omission of the corresponding self-transitions as syntactic sugar: we could agree that a UML2.0 PSM drawn by a designer is pre-processed by adding self-transitions before the PSM is formally

[14] This is sometimes done in practice by adding constraints to the states of a state diagram.

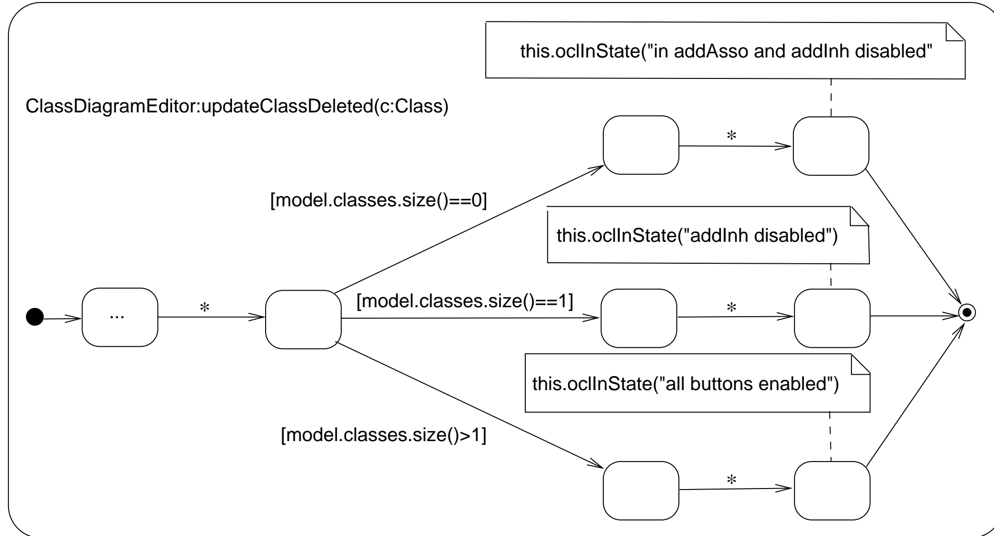

Fig. 14. Incomplete MSM for updateClassDeleted

interpreted. (This approach has the advantage of retaining the flexibility to adopt the earlier convention, which is more expressive, since it permits the designer to show that certain operations may not be legal in certain states. The UML2.0 rule cannot distinguish an operation which does not cause a state change from one which is not permitted, which seems unfortunate.)

An obvious question to ask is whether consistency is decidable. The answer depends on the choice of action language, but for any reasonable action language Turing Machines can be coded as MSMs, in which case it is easy to reduce the Halting Problem to a consistency problem, which must therefore be undecidable. Nevertheless, some tool support is possible. For example, a tool might construct representative object configurations for the different combinations of abstract states in the PSMs, symbolically execute the MSMs and check against the PSM transitions. Even if the tool's checking was not exhaustive, it might find useful counterexamples, helping the user to develop the design.

6. Incomplete method state machines

In our formalisation of the execution of MSMs we have assumed that a closed set of MSMs is given. However, it is not realistic that a designer will define an MSM for each method, even if s/he is supported by a tool. Therefore we will now briefly discuss work in progress concerning alternative ways of describing a method's behaviour.

We permit the description of a method on three levels of detail: in terms of traditional pre- and postconditions, as *incomplete MSMs*, or as executable *complete MSMs* which have been introduced in Section 5.

An incomplete MSM is an MSM fragment of which an example has already been considered above. Here we give a more precise definition: an incomplete MSM can contain *placeholder transitions*, labelled by *, and OCL constraints attached to its states. The incomplete MSM based on the MSM fragment for updateClassDeleted using this notation is given in Figure 14. Note the use of placeholder transitions, and the replacement of English constraints by OCL.

We expect that a designer will usually provide a set of method descriptions MD which consists of a mixture of pre- and postconditions, incomplete and complete MSMs. The choice of which description technique is used depends on how interesting (or difficult to design) a method appears to the designer. We do not expect this set to contain complete MSMs for all methods that are invoked in invocation boxes.

Both pre- and postconditions and incomplete MSMs are not executable but help in restricting the set of valid GSMs for the system. The complete MSMs in MD provide a partial definition of what a valid GSM looks like. The non-executable descriptions in MD provide some information about the missing parts.

As mentioned before the pre- and postconditions for a method can be represented in the PSM for the method's class. That means for each method m that is described by a pre- and postcondition in MD a GSM which is valid with respect to MD has to conform to the PSM for the class of m with this additional information.

Furthermore a valid GSM has to respect the structure specified by the incomplete MSMs in MD and fulfil their constraints. For each placeholder transition there has to be a sequence of transitions in the GSM which may include calls of other methods or creation of objects, i.e. a placeholder transition may stand for a transition sequence during which an object creation or invocation box is visited. The sequence of global transitions has to fulfil the constraints attached to the source and target states of the placeholder transition with respect to the corresponding PSM.

We do not go into more detail here, but hope to continue working on this idea in future work, especially in the context of tool support.

7. Impact of the move to UML2.0

Since we write at a time (March 2005) when UML1.5 is still the OMG's officially approved version of UML, but UML2.0 is (finally) approaching the end of the OMG's revision and adoption process, it is natural to ask whether the work will be invalidated by the adoption of the new standard. Fortunately it is clear that it will not; indeed we have been influenced by draft proposals available over the last several years.

The run-to-completion semantics of UML2.0 state diagrams has not been altered in any way which affects our discussion of the problems of recursion and callbacks, or our proposed solution. The metamodel for UML2.0 state diagrams is also not radically different from the UML1.5 metamodel used here. As here, protocol state machines and their transitions are modelled as specialisations of the more general concepts. One minor change is that in UML1.5, a Guard may be associated with each Transition, playing the role of precondition; there is no similar metamodel class to play the role of postcondition. Accordingly in this work we simply add a PostCondition metaclass to play the latter role. In UML2.0, Constraints are used for both purposes; a simple change to our metamodel would be required. As remarked earlier, whilst UML1.5 permits actions on the transitions of a protocol state machine, UML2.0 does not; we already impose the restriction, which seems to be a natural consequence of the intended use of protocol state machines.

UML2.0 radically revises activity diagrams. We have been asked whether it would be appropriate to use UML2.0 activity diagrams instead of the method state machines we propose here. Our reason for not choosing to do so is that the semantics for UML2.0 activity diagrams, while sensibly based on Petri nets, is vague in many important respects. In particular, significant work would be required to clarify how one behaviour is invoked from another (the equivalent of our invocation boxes). On reflection, it seems undesirable to try to impose the significant clarifications and restrictions on activity diagrams that would be required, because it is unclear whether the decisions we would have to make would be compatible with other intended uses of activity diagrams.

Our method state machines can be seen as a kind of UML2.0 behavioural state machines, used here to model methods. Although it is suggested (at least in UML1.5) that state machines be used to model operations, it is not explained how this is to be done. In particular, to do so precisely requires some way to specify how one operation invokes another, that permits semantically sensible use of recursion and callbacks. Our extension of state machines explains how to do this with only a small amount of extra notation, and avoids undesirable impacts on any other modelling practice.

8. Introductory example revisited

Finally we revisit the example discussed in the introduction, Figure 1. If the designer modelled with PSMs and MSMs as introduced in this paper, the state diagram would be split into a PSM and MSMs for f and g. The PSM specifies unequivocally that an object in state S1 is in state S2 after f has been executed, whatever further invocations are performed during f.

Under the assumption that the set containing the MSMs for f and g is consistent with the PSM, an object calls g during the execution of the MSM for f when it is in an appropriate state, i.e. either in S1 or in S2. According to the specification of g in the PSM, the object is either in S3 or S4 after the execution of the MSM for g has finished, depending on which state it was in at the time of g's invocation. In either case

the MSM for f has to perform further actions to guarantee that the object is in S2 after its completion, as specified in the PSM.

Variants of the notation might be considered. For example, we might permit annotation of transitions to show what callbacks were *expected* to happen. However, the designer of the class will not always be in a position to know what callbacks might happen, if other classes in the system are designed by other people. (This need not prevent the designer knowing what the state after the transition will be, given adequate contracts for called methods.)

9. Related work

We have used work by others for our definitions of PSMs and MSMs. PSMs are mentioned in UML1.5 [OMG03] and specified in more detail in UML2.0 [OMG04]. Since the UML standard must concern itself with inheritance, operations and methods are differentiated there. Further differences to our work are that PSMs are less constrained and less formally defined than here. For example all kinds of events can be attached to a ProtocolTransition in UML2.0, not only call events. The proposal also introduces additional features such as, for instance, state invariants for PSMs.

We have given a more specific and simpler definition of PSMs which is easier to handle formally but powerful enough for most practical purposes. Almost all of our well-formedness rules are described in OCL and we did not change the UML metamodel but only extended its StateMachine package. The additional features suggested in [OMG04] could be added to our model but would result in further extensions of the set of well-formedness rules and the StateMachine metamodel.

The formalism for MSMs presented in this paper is based on recursive state machines, which have been defined in [AEY01], and sequential class machines as introduced in [GLS+01]. Recursive state machines are extensions of ordinary state machines where a state can correspond to a possibly recursive invocation of a state machine. They can be used for modelling sequential imperative programs with recursive procedure calls. Besides a definition [AEY01] also contains a complexity analysis of recursive state machines concentrating on reachability and cycle detection. Similar results have been achieved independently in [BGR01].

Class machines are an object oriented extension of recursive state machines. The semantic definition of their sequential variant in [GLS+01] covers exceptions, inheritance and object creation in addition to what we have presented as MSMs. Adding a mechanism for multi-threading leads to the definition of concurrent class machines. In contrast to our work class machines are considered in isolation, not in conjunction with a more abstract modelling technique, and are not embedded into UML.

There is much work on formalisation of UML state diagrams; we only present a small subset. All of the approaches differ from ours in that they do not consider the problem of recursive calls. In [RACH00] a formalisation with labelled transition systems and algebraic specifications written in the specification language CASL is presented. Labelled transition systems are also suggested as formalism in [vdB01], where a structured operational semantics for UML state diagrams is introduced. Both in [GPP98] and in [Kus01] graph transformations are used as basis for state diagram formalisation, but these works differ in detail; [BCR02] uses ASMs. In [LMM99] state diagrams are first mapped to extended hierarchical automata and then a semantics for these specific automata is defined in terms of Kripke structures.

The application of the Observer pattern can be problematic when there are cyclic dependencies between subjects and observers or when the order of updates is of importance. In [Gru02] these two problems and their possible solutions in Java are discussed. Our approach allows a designer to model these problems and their solutions.

10. Conclusions and further work

We have pointed out that the current UML semantics for state diagrams is not sensible for situations involving recursive method calls. After showing this problem on an example we have presented an alternative approach for modelling the internal behaviour of objects using UML. In contrast to the current version of UML we differentiate between a loose specification of the effect of a method on an object and an executable machine representing an implementation of a method. We have introduced PSMs and MSMs for these purposes and defined what it means for a set of MSMs to be consistent with a PSM. Our work is fully compatible with suggestions in UML2.0 concerning the use of protocol state machines and behavioural state machines

modelling methods; it may be seen as explaining in practical detail how to do what is, in the UML2.0 standard, merely hinted at.

In future we would like to consider tool support; indeed we undertook this work because the recursive call problem prevented us from making the progress we were aiming for with work on providing tool support for the concurrent development of state and sequence diagrams. An interesting extension of MSMs would be to add inheritance, which as briefly mentioned in Section 5 would raise both theoretical questions and issues in practical modelling.

Acknowledgements We are grateful to the British Engineering and Physical Sciences Research Council for funding (GR/N13999/01, GR/A01756/01). We would also like to thank the FAC referees for their helpful and constructive comments.

References

[AEY01] R. Alur, K. Etessami, and M. Yannakakis. Analysis of recursive state machines. In *Proceedings of Computer Aided Verification, CAV'01*, volume 2102 of *LNCS*, pages 207–220. Springer, 2001.

[Arg] The ArgoUML Project. Website at http://argouml.tigris.org/.

[BCR02] E. Börger, A. Cavarra, and E. Riccobene. A precise semantics of UML state machines: making semantic variation points and ambiguities explicit. In *Proceedings of Semantic Foundations of Engineering Design Languages, SFEDL, Satellite Workshop of ETAPS'02*, 2002.

[BGR01] M. Benedikt, P. Godefroid, and T.W. Reps. Model checking of unrestricted hierarchical state machines. In *Proceedings of Automata, Languages and Programming, ICALP'01*, volume 2076 of *LNCS*, pages 652–666. Springer, 2001.

[BRJ98] Grady Booch, James Rumbaugh, and Ivar Jacobson. *The Unified Modeling Language User Guide*. Addison Wesley Longman, 1998.

[GHJV95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, 1995.

[GLS+01] R. Grosu, Yanhon A. Liu, S.A. Smolka, S.D. Stoller, and J.Yan. Automated software engineering using concurrent class machines. In *Proceedings of Automated Software Engineering, ASE'01*. IEEE, 2001.

[GPP98] M. Gogolla and F. Parisi-Presicce. State diagrams in UML: A formal semantics using graph transformations. In *Proceedings of the Workshop on Precise Semantics for Modeling Techniques, PSMT'98*. Technische Universität München, TUM-I9803, 1998.

[Gra98] Mark Grand. *Patterns in Java*, volume 1. John Wiley & Sons, 1998.

[Gru02] D. Gruntz. Java design: On the observer pattern. *Java Report*, February 2002.

[HG97] D. Harel and E. Gery. Executable object modeling with statecharts. *IEEE Computer*, 30:7:31–42, 1997.

[Kus01] S. Kuske. A formal semantics of UML state machines based on structured graph transformation. In *Proceedings of the 4th International Conference on the Unified Modeling Language, UML'01*, volume 2185 of *LNCS*, pages 241–256. Springer, 2001.

[LMM99] D. Latella, I. Majzik, and M. Massink. Towards a formal operational semantics of UML statechart diagrams. In *Proceedings of Formal Methods for Open Object-Based Distributed Systems, FMOODS'99*, volume 139 of *IFIP*. Kluwer, 1999.

[MSL99] L. Mikhajlov, E. Sekerinski, and L. Laibinis. Developing components in the presence of re–entrance. In *Proceedings of Formal Methods – FM'99*, volume 1709 of *LNCS*, pages 1301–1320. Springer, 1999.

[OMG03] Object Management Group OMG. *Unified Modeling Language Specification version 1.5*, March 2003. Available from http://www.omg.org/uml/ as formal/03-03-01.

[OMG04] Object Management Group OMG. *UML2.0 Superstructure Specification*, October 2004. Available from http://www.omg.org/uml/ as ptc/04-10-02.

[Par03] U2 Partners. Unified Modeling Language 2.0 proposal, version 2.0, April 2003. Available from http://www.u2-partners.org/.

[RACH00] G. Reggio, E. Astesiano, C. Choppy, and H. Hussmann. Analysing UML active classes and associated state machines — A lightweight formal approach. In *Proceedings of Fundamental Approaches to Software Engineering, FASE'00*, volume 1783 of *LNCS*, pages 127–146. Springer, 2000.

[Szy98] Clemens Szyperski. *Component Software: Beyond Object-Oriented Programming*. ACM Press books. Addison Wesley, 1998.

[TS03] Jennifer Tenzer and Perdita Stevens. Modelling recursive calls with UML state diagrams. In *Proc. Fundamental Approaches to Software Engineering*, number 2621 in LNCS, pages 135–149. Springer-Verlag, April 2003.

[vdB01] M. von der Beeck. Formalization of UML-Statecharts. In *Proceedings of the 4th International Conference on the Unified Modeling Language, UML'01*, volume 2185 of *LNCS*, pages 406–421. Springer, 2001.

[WK99] J. Warmer and A. Kleppe. *The Object Constraint Language: Precise Modeling with UML*. Addison Wesley, 1999.

[wRPon] Perdita Stevens with Rob Pooley. *Using UML: Software Engineering with Objects and Components*. Addison Wesley Longman, 1999 (updated edition).